Imposition of a Relaxed No Slip Condition in PINNs

Ameir Shaa

November 29, 2023

Table of Contents

Relaxed No Slip Boundary Conditions

Introduction

- Layton [**?**] introduces penalty methods to replace the Lagrange multiplier imposition of the constraint $\mathbf{u} \cdot \hat{\tau}|_{\Gamma} = 0$.
- This constraint is in relation to the Stokes problem -

$$\begin{aligned} -\Delta \mathbf{u} + \nabla p &= \mathbf{f}, && \text{in } \Omega \left(\subset \mathbb{R}^2\right) \\ \mathbf{u} &= 0, && \text{on } \partial\Omega \end{aligned}$$

- where $\mathbf{f}$ is the external force, p is the pressure, and $\mathbf{u}$ is the velocity.
- The Stokes problem is a fundamental partial differential equation that describes fluid flow.
- It operates within a domain Ω in a two-dimensional space, $\mathbb{R}^2$, with specific boundary conditions where the fluid velocity $\mathbf{u}$ is zero on the boundary $\partial\Omega$.

- Layton proposes a relaxed form of the boundary condition where $(\mathbf{u} \cdot \hat{\tau}, \alpha)_\Gamma = 0$.
- This is the relaxed no-slip boundary condition, where $\mathbf{u} \cdot \hat{\tau}$ should be zero on the boundary Γ and $\mathbf{u} \cdot \hat{n}$ is zero on the boundary.
- In other words, the tangential velocity on the boundary is relaxed to some extent by a parameter α and the normal velocity on the boundary is set to zero.

- Layton [**?**] discusses several techniques for relaxing the no slip boundary condition but we will look at the concept of penalty functions and regularization.
- Consider the following penalized functional:

$$J_\epsilon(\mathbf{v}) = \int_\Omega \frac{1}{2}|\nabla \mathbf{v}|^2 - \mathbf{f}\cdot\mathbf{v}\, dx + \frac{1}{2}\epsilon^{-1}\int_\Gamma (\mathbf{v}\cdot\hat{\tau})^2\, ds \tag{1}$$

  where $\epsilon > 0$.
- This functional is used to find $\mathbf{u}_\epsilon \in V$ that minimizes $J_\epsilon(\mathbf{v})$ where V is a function space defined as
  $V := \left\{ \mathrm{v} \in X : (\nabla\cdot \mathrm{v}, q) = 0, \forall q \in L_0^2(\Omega) \right\}$.
- The given functional $J_\epsilon(\mathbf{v})$ represents a penalized form commonly used in variational problems, particularly in fluid dynamics or elasticity.

- Let us discuss the terms present in greater detail.
  1. **First Term:** $\int_\Omega \frac{1}{2}|\nabla \mathbf{v}|^2 \, dx$ - This term is an integral over the domain Ω. $|\nabla \mathbf{v}|^2$ represents the square of the gradient norm of the vector field $\mathbf{v}$, typically corresponding to kinetic or strain energy. It reflects the energy associated with the spatial variation of $\mathbf{v}$. For our purposes, since we take velocities inside the building to be zero, this term is then necessarily zero.
  2. **Second Term:** $-\int_\Omega \mathbf{f} \cdot \mathbf{v} \, dx$ - This integral is over Ω. $\mathbf{f} \cdot \mathbf{v}$ is the dot product of a field $\mathbf{f}$ and the vector field $\mathbf{v}$. It accounts for the work done by the field $\mathbf{f}$ against $\mathbf{v}$. For our purposes, since we do not consider any external forces and the velocity is zero inside our buildings, this term is zero as well.
  3. **Third Term:** $\frac{1}{2}\epsilon^{-1} \int_\Gamma (\mathbf{v} \cdot \hat{\tau})^2 \, ds$ - This integral is over the boundary Γ. $(\mathbf{v} \cdot \hat{\tau})^2$ is the square of the tangential component of $\mathbf{v}$ along the boundary. ϵ is a small positive parameter, making this a penalization term. It penalizes the tangential component of $\mathbf{v}$ at the boundary, enforcing a boundary condition. This is the only term that contributes to our relaxed no slip boundary condition.

- In summary, Equation 1 is now simplified to

$$J_\epsilon(\mathbf{v}) = \frac{1}{2}\epsilon^{-1}\int_\Gamma (\mathbf{v}\cdot\hat{\tau})^2\, ds \tag{2}$$

- where $\epsilon > 0$.
- The functional $J_\epsilon(\mathbf{v})$ is used in variational problems to find a vector field $\mathbf{v}$ that minimizes this functional.
- The penalization term enforces boundary conditions by penalizing certain behaviors at the boundary.
- The choice of ϵ is crucial for balancing boundary condition enforcement and numerical stability.
- $J_\epsilon(\mathbf{v})$ combines energy terms in the domain with a penalization term at the boundary.
- The goal is to find a vector field $\mathbf{v}$ that minimizes this functional while respecting boundary conditions enforced through the penalization term.

Table of Contents

- To begin, we consider what our geometries look like.
- Each simulation comes with a specific STEP (Standard for the Exchange of Product Data) file.
- We create a mesh and export it.
- Then we later import the data as a set of points that we can work with.

```
import FreeCAD
import Mesh

def mesh_step_to_def(step_file_path, output_file_path):
    # Load the STEP file
    doc = FreeCAD.newDocument()
    FreeCAD.open(step_file_path)
    FreeCAD.setActiveDocument(doc.Name)

    # Assuming the STEP file has only one object
    obj = doc.Objects[0]

    # Mesh the object
    mesh = Mesh.Mesh()
    mesh.addMesh(obj.Mesh)
```

```
# Prepare the mesh data for export
# This part depends on how you want to structure your DEF
    file
mesh_data = {
    "vertices": [(v.Point.x, v.Point.y, v.Point.z) for v in
        mesh.Topology[0]],
    "facets": mesh.Topology[1]
}

# Export the mesh data to a Python DEF file
with open(def_file_path, 'w') as file:
    file.write(str(mesh_data))
```

- The structure of the output mesh is such that each entry in your list represents the coordinates of a single vertex of a triangular face.
- For a complete face, one would need sets of three such entries.
- To compute normals and perform other tasks, one needs to reshape this data into a format where each face is defined by three vertices.

```
def restructure_data(geometry_module):

from geometry_module import faces as faces_data

# Reshaping into faces
faces = [tuple(faces_data[i:i+3]) for i in range(0,
    len(faces_data), 3)]

return faces
```

- The ability to extract the normal to the defined geometries at any point on the surface of the geometry is crucial to our boundary condition.
- Here we develop a method to do so.

```python
import numpy as np

def compute_normal(v1, v2, v3):
    # Convert to numpy arrays
    v1, v2, v3 = np.array(v1), np.array(v2), np.array(v3)
    # Compute edges
    edge1 = v2 - v1
    edge2 = v3 - v1
    # Compute normal
    normal = np.cross(edge1, edge2)
    # Normalize
    normal = normal / np.linalg.norm(normal)
    return normal

# Example usage
normals = [compute_normal(*face) for face in subfaces]
```

- This function enforces the relaxed no-slip boundary condition on the sphere and cylinder surfaces.
- It samples points from both surfaces and assigns zero velocities to them, representing the no-slip condition.
- This function returns the (scaled) points on the surface of the geometry, the normal unit vectors corresponding to the points on that surface and the no-slip (scaled) velocities.

```
def no_slip_boundary_conditions(device, faces_data, num_points,
    wind_angle, feature_scaler, target_scaler):
    """
    Returns the sampled points on the geometry surface along
        with their corresponding no-slip velocities and the
        normals.

    Parameters:
    - geometry: List of points on the surface of the geometry
    - num_points: Percentage of points to sample on the geometry
        (0,1]
    - wind_angle: Angle of the inlet wind direction

    Returns:
    - no_slip_points: Coordinates of the sampled points on the
        geometry's surface in the representation expected by the
        NN.
    - no_slip_velocities: Velocities (all zeros) at the sampled
        points, representing the no-slip condition.
    - no_slip_normals: Normal vectors corresponding to the
        coordinates of the sampled points on the geometry's
```

```
faces = restructure_data(faces_data)
subfaces = random.sample(faces, int(len(faces) * num_points))
no_slip_normals = [compute_normal(*face) for face in
    subfaces]

# Flatten the list of faces into a list of points
flattened_points = [point for face in faces for point in
    face]

# Separate x, y, and z coordinates
x_coords = [point[0] for point in flattened_points]
y_coords = [point[1] for point in flattened_points]
z_coords = [point[2] for point in flattened_points]

# Convert to NumPy arrays
x_array = np.array(x_coords)
y_array = np.array(y_coords)
z_array = np.array(z_coords)
```

```
# Create the DataFrame
df_features = pd.DataFrame({
    'Points:0': x_array,
    'Points:1': y_array,
    'Points:2': z_array
})

# Add cos(WindAngle) and sin(WindAngle) as new columns
df_features['cos(WindAngle)'] =
    np.cos(np.deg2rad(wind_angle))
df_features['sin(WindAngle)'] =
    np.sin(np.deg2rad(wind_angle))

df_targets = pd.DataFrame(index=range(len(df_features)))

# Add zero velocities
df_targets['Velocity:0'] = 0.0
df_targets['Velocity:1'] = 0.0
df_targets['Velocity:2'] = 0.0
```

```
    # Scale the features accordingly to the distribution of the
        original dataset
    normalized_features, normalized_targets =
        transform_data_with_scalers(df_features, df_targets,
        feature_scaler, target_scaler)

    # Convert to Torch tensor
    no_slip_points = torch.tensor(normalized_features,
        dtype=torch.float32)
    no_slip_velocities = torch.tensor(normalized_targets,
        dtype=torch.float32)

    no_slip_points = no_slip_points.to(device)
    no_slip_velocities = no_slip_velocities.to(device)

    return no_slip_points, no_slip_velocities, no_slip_normals
```

```
def compute_no_slip_loss(self, X_boundary, normals, epsilon,
    output_params):

    def find_velocity_indices(output_params):
        velocity_labels = ['Velocity:0', 'Velocity:1',
            'Velocity:2']
        velocity_indices = [output_params.index(label) for
            label in velocity_labels]
        return velocity_indices

    def compute_tangential_velocity(velocities, normals):
        normal_velocities = torch.sum(velocities * normals,
            dim=1, keepdim=True) * normals
        tangential_velocities = velocities - normal_velocities
        return tangential_velocities, normal_velocities

    def compute_penalty_term(tangential_velocities, epsilon):
        penalty = torch.mean(tangential_velocities**2) #
            (( v )  ds) term
        penalty = penalty / (2*epsilon)
        return penalty
```

```
velocity_indices = find_velocity_indices(output_params)
start_idx, end_idx = min(velocity_indices),
    max(velocity_indices) + 1

boundary_predictions = self(X_boundary)
velocities = boundary_predictions[:, start_idx:end_idx]

tangential_velocities, normal_velocities =
    compute_tangential_velocity(velocities, normals)
penalty = compute_penalty_term(tangential_velocities,
    epsilon)

criterion = nn.MSELoss()
normal_loss = criterion(normal_velocities,
    torch.zeros_like(normal_velocities))

no_slip_loss = penalty + normal_loss

return no_slip_loss
```